Certificación PCAP

Lección 3: Listas, tuplas y diccionarios

eip ESCUELA INTERNACIONAL POSGRADOS

ÍNDICE

Lección 3: Listas, tuplas y diccionarios

1. Objetivos

En esta lección continuaremos con el repaso de los aspectos más relevantes del contenido cubierto por el PCAP. Veremos primeramente las principales características de los objetos de tipo “lista” como cuál es la manera de definirlas en Python, qué propiedades tienen y cómo se puede operar con ellas.

Seguidamente, pasaremos a la revisión de sus hermanas las tuplas, con aspectos similares a las listas, pero con sus propias particularidades.

Veremos también otro tipo de variable muy recurrente, los diccionarios. Estudiaremos, igualmente sus características y las principales operaciones que se pueden hacer con ellos.

Nuestro objetivo, nuevamente, en esta lección no será tanto el estudio en profundidad de cada uno de los apartados de la misma, sino que más bien, intentaremos enfocarnos en los aspectos y/o detalles que con más probabilidad pueden aparecer en el examen de la certificación. Principalmente en los detalles que podemos pasar por alto y que pueden pasarnos factura en dicho examen.

2. Listas

2.1. Definición y características principales

Las listas son la respuesta natural al crecimiento de la cantidad de datos que se necesita almacenar, o con los que se desea operar; cuando tener una variable por cada uno de los valores que podamos necesitar en nuestro código no resulte ágil, ni útil. Aun así, no quiere decir esto que las listas solo sean útiles en entornos donde haya muchos datos que tratar, sino que en desarrollos más simples pueden ser de gran ayuda, simplificando la cantidad de trabajo que sería necesario si estas no existieran.

La definición más general de **listas** en Python es una **colección de datos ordenada** y se declaran de la siguiente manera

mi_lista = [valor1, valor2, valor3.... valorN]

```
1.      >>> mi_lista = [1, "a", 3, "datos", True]
```

La asignación de las listas a variables en Python se hace principalmente escribiendo los valores contenidos en dicha lista separados por comas y entre corchetes "[]". Estas listas pueden estar compuestas por cualquier número de elementos.

Una de las primeras observaciones que podemos hacer viendo el ejemplo es la enorme "simpleza" con la que este tipo de variables se declara en Python en comparación con la definición realizada en otros lenguajes como por ejemplo java, del cual dejamos el ejemplo a continuación.

```
1. List<Integer> miLista = Arrays.asList(3, 1, 4);
```

Además, comparando las dos definiciones podremos observar otra pequeña (o gran) diferencia.

En la lista definida en java, se ha de definir el tipo de datos que contendrá la lista antes de crearla, y todos estos elementos (en el caso del ejemplo), tienen que ser de tipo *integer* o entero. No así en el caso de Python, donde **sí es posible realizar combinaciones de diferentes tipos de datos en la misma definición de la lista** (enteros, cadenas, booleanos, arrays, etc).

Esta característica no es exclusiva de Python, pero nuevamente, hace destacar a este lenguaje en el aspecto de sencillez y flexibilidad otorgándole una gran versatilidad susceptible de ser aprovechada en casi cualquier desarrollo que se pretenda realizar.

2.2. Manipulación y operaciones con listas

Según lo visto, las listas tienen mucha utilidad como almacenadoras de datos, pero las ventajas que ofrecen no solo se limitan a esto, sino que otra de sus grandes virtudes es la capacidad de alterarlas y operar con los valores que estas contienen. Es por ello que en los siguientes apartados hablaremos de las diferentes maneras que tenemos tanto para leer como para manipular su contenido.

Función len. Longitud de una lista

Las listas pueden contener cualquier cantidad de elementos y a su vez, esta cantidad puede ser alterada como veremos posteriormente. Ante esta realidad puede resultar útil en muchos casos conocer el número de elementos que componen las listas con las que estamos trabajando. Python provee una solución a través de la función "**len**". A continuación, la sintaxis para su utilización

len(mi_lista)

Esta función toma como argumento la lista de la cual queremos conocer su longitud y devuelve como resultado dicha longitud.

```
1.      >>> mi_lista = [1,2,3,4,5,6]
2.      >>> print(len(mi_lista))
3.      6
```

Indexación. Utilización de índices negativos

Otra de las utilidades que tienen las listas es la posibilidad de leer o escoger valores de entre todos los almacenados en su interior. Para esto, se asigna un **índice** a cada una de las posiciones de los elementos que conforman la lista. El primer elemento, por convenio, tiene el índice cero (**0**), el segundo, el uno (1) y asi sucesivamente. La acción mediante la cual se accede a un elemento de la lista se denomina indexación y la sintáxis para ello es la que encontramos a continuación.

elemento_seleccionado = mi_lista[índice]

Indexando una lista real:

```
>>> mi_lista = [1, "dos", True, 7]

>>> mi_lista[0]
1
>>> mi_lista[1]
'dos'
>>> mi_lista[2]
True
>>> mi_lista[3]
 7
```

Pero no solo podemos utilizar valores enteros para indexar una lista, sino que también podremos utilizar operaciones. Para ello, el resultado que arrojen estas operaciones debe cumplir dos condiciones:

- Que el resultado de la operación sea un entero. Si el resultado es un float la indexación fallará, y obtendremos un mensaje de error indicándolo.

- Una vez que se cumpla la anterior condición, lo siguiente es que el resultado tenga algún sentido, dicho en otras palabras, que ese número represente a un elemento existente de la lista. Si la lista tiene 3 elementos y queremos acceder al índice 10, el cual no existe, obtendremos un error.

Pasemos a verlo con algunos ejemplos.

```
>>> mi_lista= [1, 2, 3, 4, 5, 6]
>>> print(mi_lista[2+2]) # Índice como resultado de
                         # una suma
5
>>> print(mi_lista[5//2]) # Índice como resultado de
                          # una división entera
3
>>> print(mi_lista[5%2])   # Índice como resultado de
                          # un módulo
2

>>> print(mi_lista[5/2])   # Índice de tipo float
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: list indices must be
    integers or slices, not float

>>> print(mi_lista[5+5]) # Índice fuera de rango
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range

>>> print(mi_lista[True]) # Pasamos un bool como
                         # índice
2
>>> print(mi_lista[False]) # Pasamos un bool como
                          # índice
1
>>> print(1==True)
True
>>> print(0==False)
True

```

Un caso particular, y que merece la pena recalcar nuevamente, son los dos últimos ejemplos, donde vemos que pasamos los valores booleanos “True” y “False” en la indexación. Puede que el resultado nos sorprenda. Para Python 1 == True y 0 == False, con lo cual, intentar acceder al índice True, Python asume que hablamos del índice 1, lo mismo para False.

En ambos casos recuperaremos un valor, siempre que este exista.

Llegados aquí, podemos plantearnos lo siguiente, ¿y si el resultado de la operación que utilizamos en la indexación es negativo?

Pues bien, en Python también es posible utilizar índices negativos para leer el valor de un elemento de la lista, donde el -1 representa el índice del último elemento de la misma, -2 el penúltimo y así sucesivamente.

Esto puede resultar un tanto extraño al principio, pero puede ser también muy útil en ciertas situaciones ,como por ejemplo, si quisiéramos leer el último elemento de una lista de la cual no conocemos su longitud.

La utilización de los índices negativos debe cumplir con las condiciones mencionadas anteriormente, i.e. el resultado tiene que ser un entero (no puede ser un float) y el índice debe tener sentido y debe representar a un elemento existente.

De la misma manera que intentar acceder a un índice mayor que la longitud de la lista menos 1 arroja un error, intentar lo mismo en la parte negativa arrojará el mismo error. Es decir, las cotas superior e inferior para los índices accesibles de una lista pueden resumirse en la siguiente expresión.

-len(mi_lista) <= índice <= len(mi_lista)-1

Donde la corrección "-1" de la parte positiva viene por el hecho de que el primer elemento de las listas tiene el índice 0.
Podemos verlo más claro a continuación

```
>>> mi_lista = [1, "dos", True, 7]
>>> mi_lista[-1]
7
>>> mi_lista[-2]
True
>>> mi_lista[-3]
'dos'
>>> mi_lista[-4]
1
>>> mi_lista[-5]
Traceback (most recent call last):
   File "<stdin>", line 1, in <module>
IndexError: list index out of range

>>> mi_lista[3]
7

>>> mi_lista[4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range

```

Dónde observamos claramente que el índice inferior que podemos utilizar para indexar es el -4, que representa al primer elemento y el mayor es el 3, que representa al último.

Añadir nuevos elementos a la lista

Al trabajar con listas es muy probable que sea necesario alterar el contenido de estas, en particular, puede que en determinadas circunstancias sea necesario añadir nuevos elementos al final de la lista. Para añadir nuevos elementos al final de la lista python proporciona el método "**append()**".

La sintaxis para la utilización de esta instrucción es la siguiente

mi_lista.append(valor)

Utilizando esta instrucción con una lista real en un ejemplo práctico, tenemos lo siguiente:

```
1.      >>> mi_lista = [1, "dos", 3, "4”]
2.      >>> mi_lista.append('cadena de texto')
3.      >>> print(mi_lista)
4.      [1, 3, '4', 'cadena de texto']
5.
```

Donde podemos observar claramente como utilizando la instrucción **append** somos capaces de añadir un nuevo elemento al final de la lista.

Eliminar elementos de la lista. Instrucción "del"

Además, puede que en determinadas circunstancias sea necesario eliminar determinados elementos que ya no son necesarios para el desarrollo. En estos casos se utiliza la palabra reservada "**del**", una instrucción de python cuya utilidad nos permite borrar elementos de las listas.

La sintaxis para la utilización de esta instrucción es la siguiente

del mi_lista[índice]

Utilizando esta instrucción con una lista real en un ejemplo práctico, tenemos lo siguiente:

```
>>> mi_lista = [1, "dos", 3, "4", True,
                "cadena de texto", False]
>>> del mi_lista[1] # Eliminamos el elemento
>>> print(mi_lista)
[1, 3, '4', True, 'cadena de texto', False]
```

Donde podemos observar claramente como utilizando la instrucción **del** somos capaces de eliminar el elemento que deseemos. Debemos tener en cuenta que, ante esta situación, conviene tener en cuenta que eliminar elementos puede llevar, en algunos casos y si no se trabaja con cuidado, a la aparición de errores. Son dos los aspectos que tenemos que considerar al utilizar "del":

- El primero es que al eliminar un elemento de una lista, alteramos su longitud, y consecuentemente, el índice máximo al cual podremos acceder ya no será el mismo que antes de borrar.

  A continuación, un ejemplo del error que obtendríamos en este supuesto.

```
>>> mi_lista = [1,2,3,4,5]

>>> del mi_lista[2]

>>> print(mi_lista[4])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

- El segundo, es que los índices de los elementos de la lista cuyos índices son mayores que el del elemento eliminado se verán alterados. En palabras es menos claro, con el ejemplo se explica mucho mejor.

```
>>> mi_lista = [1,2,3,4,5]

>>> print(mi_lista[3])
4
>>> del mi_lista[2]

>>> print(mi_lista[3])
5
```

El elemento con índice 3 antes de borrar era el entero 4, pero después de borrar el elemento con índice 2, el índice 3 ahora está asignado al entero 5.
Finalmente, en relación a esta instrucción, también tenemos la posibilidad de borrar por completo la lista y no solo alguno de sus elementos. Para ello no tenemos más que escribir lo siguiente:

del mi_lista

Debemos tener cuidado al ejecutar esta instrucción, ya que a partir del momento de su ejecución, intentar acceder a cualquier elemento de la misma arrojará un error dado que la lista ya no será reconocible como variable.

```
>>> mi_lista = [1,2,3,4,5]
>>> del mi_lista
>>> mi_lista[2]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'mi_lista' is not defined

```

Actualizando el contenido de una lista

Otra de las circunstancias más frecuentes a la hora de trabajar con listas es la necesidad de actualizar los valores de los datos que contiene. En python podemos hacerlo mediante la indexación y el operador de asignación "=". De tal manera que si queremos cambiar el valor de un elemento, debemos primeramente conocer el índice de dicho elemento y aplicar lo siguiente:

mi_lista[indice] = nuevo_valor

```
>>> mi_lista = [1, "dos", True, False, 5]
>>> mi_lista[2] = "nuevo valor para este elemento"
>>> mi_lista
[1, 'dos',
'nuevo valor para este elemento', False, 5]
>>>
```

Vemos como antes de cambiar el elemento de índice 2 su valor era un bool "True", y después se modifica a un string. Esto es totalmente posible en las listas de Python, al actualizar un elemento, no es necesario que el nuevo valor sea del mismo tipo que el antiguo, podemos cambiarlo a nuestra conveniencia. Al actualizar el contenido de la lista hay que tener cuidado ya que si se intenta actualizar el contenido de una posición que no existe devolverá error.

```
>>> mi_lista = [1, "dos", True, False, 5]
>>> mi_lista[5] = "nuevo valor para este elemento"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list assignment index out of range

```

Seleccionando elementos de una lista: *Slices*

¿Qué hacemos cuando tenemos una lista con valores y queremos seleccionar o copiar solo una parte de todos ellos? La respuesta a esta situación son los slices o "rebanadas" como se conocen por su traducción. Estas "rebanadas" son instrucciones que nos permiten realizar copias de las listas seleccionando qué elementos de estas son los que nos interesan.

La sintáxis para los slices es la siguiente:

mi_lista[A:B]

Donde el primer valor (A) representa el índice del primer elemento que se seleccionara y el segundo valor (B) representa **el primero que NO se incluirá**, tal y como podemos ver el en ejemplo siguiente:

```
>>> mi_lista = [1,2,3,4,5]
>>> print(mi_lista[1:4])
[2, 3, 4]
>>>
```

El elemento de indice 4 (mi_lista[4] = 5) no está incluido en la selección. **Conviene tener muy presente siempre esta característica, ya que es muy recurrente en el examen.**

Es posible hacer slices de las listas simplificando un poco más la sintáxis. Supongamos que nos encontramos en una de las siguientes situaciones:

- Nos interesa seleccionar todos los elementos de la lista desde el elemento inicial hasta un índice determinado.
- Nos interesa seleccionar todos los elementos de la lista desde un índice determinado hasta el final de la misma.

En estos casos, la sintaxis de la rebanada se puede simplificar de la manera que podemos ver en el siguiente ejemplo:

```
>>> mi_lista = [1,2,3,4,5]
>>> print(mi_lista[:4])
 [1, 2, 3, 4]

>>> print(mi_lista[2:])
 [3, 4, 5]

```

Simplemente, si omitimos el índice inicial o el final, le estaremos diciendo a python que queremos que seleccione los elementos de la lista desde el principio o hasta el final respectivamente.

De la misma manera, también se puede complicar la selección de elementos de la lista. Supongamos ahora que nos encontramos en una situación en la que nos interesa seleccionar un elemento cada “N” elementos de la lista original, puede que no sea muy corriente esta situación, o quizás sí, lo importante es que igualmente python provee de una manera de hacerlo a través de la siguiente instrucción

mi_lista[A:B:N]

Donde A representa el índice inicial, B, el índice final y N el "paso de salto" de esa selección. Lo vemos en acción en el siguiente ejemplo.

```
1.      >>> mi_lista = [1,2,3,4,5,6,7,8,9]
2.      >>> print(mi_lista[0:8:2])
3.       [1, 3, 5, 7]
4.
```

Elementos seleccionados de la lista **mi_lista**
mi_lista[0] = 1
mi_lista[0+N] = mi_lista[2] = 3
mi_lista[2+N] = mi_lista[4] = 5
mi_lista[4+N] = mi_lista[6] = 7

Tabla 2.1. Tabla explicativa del procedimiento de slice realizado por Python

Nuevo detalle a tener en cuenta. Mientras que en el caso del parámetro **B**, el valor que tome este representa el índice del primer elemento que **NO** se seleccionará en el slice, el comportamiento de la **N** es un poco distinto, ya que su valor representa cual es el primer indice que **SÍ** se tomará contando a partir del anterior. En palabras más simples y utilizando el ejemplo del slice anterior (mi_lista[0:8:2]); mientras que el índice 8 no formará parte de la selección, el índice 0+N = 2 sí forma parte de la selección. Pequeñas sutilezas que merece la pena tener en mente, al menos de cara al examen del PCAP.

En realidad los siguientes slices con equivalentes y devolverán el mismo resultado

mi_lista[A:B] == mi_lista[A:B:1]

Cuando se omite el valor del paso, N por defecto viene definido a 1.

Las rebanadas también pueden combinarse con el uso de los índices negativos sin ningún tipo de problema, siempre y cuando respetemos la regla principal de que el elemento representado por el índice **A** siempre debe estar situado antes que el elemento representado por el índice **B** (no siempre, existe una excepción dependiendo del valor de N. Lo veremos a continuación), **si esto no se cumple, el resultado será una lista vacía**. Observamos estas características a continuación.

```
>>> mi_lista = [1,2,3,4,5,6,7,8,9]

>>> print(mi_lista[1:-1]) # El último índice es
                          # negativo
[2, 3, 4, 5, 6, 7, 8]

>>> print(mi_lista[1:-1:2]) # El índice negativo es
                         # válido en rebanadas con paso
[2, 4, 6, 8]

>>> print(mi_lista[-8:-1:3]) # Ambos negativos también
                          # es válido
[2, 5, 8]

>>> print(mi_lista[4:1]) # Si A es posterior a B,
                         # resultado vacío
[]

```

Finalmente, hablamos de una posibilidad más que ofrece el uso de los slices en python, ya que al igual que tanto el índice inicial como el final pueden ser enteros negativos (u operaciones cuyos resultados sean enteros que representen un valor existente) también podemos utilizar como entero negativo el valor del paso de los slices. Es decir, podremos encontrarnos con un "N" negativo. Veámoslo en acción.

```
>>> mi_lista= [1, 2, 3, 4, 5, 6]
>>> print(mi_lista[0:5:-1])
[]
>>> print(mi_lista[-1:0:-1])
[6, 5, 4, 3, 2]

```

En dicha situación observamos como se invierte la condición de que el índice A debe representar a un elemento que aparezca antes que aquel

representado por el índice B, todo ello debido a que N es negativo. Mientras que mi_lista[0:5:-1] devuelve una lista vacía, mi_lista[-1:0:-1] no.

Podemos resumir este comportamiento como sigue

	mi_lista[A] antes que mi_lista[B]	mi_lista[B] antes que mi_lista[A]
if N > 0	Lista no vacía	Lista vacía
if N < 0	Lista vacía	Lista no vacía

Tabla 2.2. Resumen del comportamiento de los slices dependiendo del valor del paso N

En la siguiente tabla ejemplificaremos el comportamiento de los slices para N>0 y N<0 en la lista de ejemplo mi_lista =[1,2,3,4,5,6,7].

mi_lista[0:-1:2]	mi_lista[-1:1:-2]
mi_lista[0] = 1	mi_lista[-1] = 7
mi_lista[0+N] = mi_lista[2] = 3	mi_lista[-1+N] = mi_lista[-3] = 5
mi_lista[2+N] = mi_lista[4] = 5	mi_lista[-3+N] = mi_lista[-5] = 3
mi_lista[4+N] = mi_lista[6] = 7 **NO SE INCLUYE**	mi_lista[-5-N] = mi_lista[-7] = 1 **NO SE INCLUYE**

Tabla 2.3. Ejemplo del comportamiento del slice con un N > 0 vs uno con N < 0

NOTA. Añadimos el siguiente ejemplo de slice de una lista que es muy típico que aparezca en alguna de las preguntas del examen. Si hemos entendido correctamente lo visto hasta ahora no tendremos problemas en entender el resultado del mismo.

```
1.      >>> mi_lista = [1,2,3,4,5]
2.      >>> print(mi_lista[-5:4])
3.       [1, 2, 3, 4]
4.
```

Hacer un print desde esta manera no recorre 2 veces la lista. Es decir, al hacer mi_lista[-5:4] en ningún caso obtendremos [1,2,3,4,5,1,2,3,4], que sería el resultado de leer del elemento -5 al -1 y posteriormente del 0 al 4. Los índices

que se introducen en el slice no tienen relación entre ellos mismos, solo representan elementos de la lista y cada uno se encuentra en una posición determinada. El slice comenzará en el primero y continuará hasta encontrar el segundo, nada más.

Los slices son muy importantes dentro del tratamiento con listas, ya que su utilización se puede combinar con otras instrucciones y funciones, como por ejemplo con la instrucción "del" vista anteriormente.

De esta manera, ya no solamente podremos borrar un elemento determinado de la lista, si no que podemos utilizar los slices como sigue:

del mi_lista[A:B:N]

La sentencia anterior eliminará de la lista "mi_lista" desde el elemento con índice A, hasta el elemento con índice B (no incluido) en saltos de N. Como podremos imaginar, todas las reglas y casuísticas vistas en los slices son aplicables ahora.

```
1.      >>> mi_lista = [1, 2, 3, 4, 5, 6, 7]
2.
3.      >>> del mi_lista[-1:2:-1]
4.      >>> mi_lista
5.      [1, 2, 3]
6.
```

Mutabilidad de los objetos de tipo lista

Llegados a este punto, una vez que ya hemos hablado de la instrucción "del", de los slices y de la actualización de datos de una lista, es necesario que hablemos sobre una de las principales características diferenciadoras de las listas y de otras variables complejas de python, donde todos estos elementos serán de gran utilidad.

Esta característica tiene que ver con cómo realiza python la asignación a las variables y qué significado y comportamiento tiene esta asignación ante la alteración de dichas variables.

Como seguramente hayamos escuchado más de una vez, Python es un lenguaje donde todo son "objetos". Es decir, cuando realizamos una asignación de un valor a una variable como la siguiente:

```
1.      >>> a = 3
2.
```

lo que python hace es crear el objeto entero 3 y **asignar** (no es una igualdad matemática) el nombre "a" a ese objeto para poder tratar con el posteriormente hasta que la ejecución del programa termine. De la misma manera ocurre con todas las variables que podemos definir, sean listas, booleanos, strings, etc.

Ahora bien, los objetos, en Python pueden ser divididos en dos grandes grupos, objetos **inmutables** (que no se pueden modificar) y objetos **mutables** (sí se pueden modificar). Los enteros, las cadenas o las tuplas, son ejemplos de objetos inmutables, mientras que las listas y los diccionarios, por el contra, sí que lo son.

Como bien podremos imaginar, las diferencias en el comportamiento entre variables mutables e inmutables se observará claramente en el momento de intentar modificar las mismas. Además, para explicar mejor este concepto vamos a introducir una nueva función de python, la **función id()**. Esta función devuelve la dirección de memoria del objeto dado.

Veamos como se comporta una variable inmutable cuando se intenta alterar

```
1.      >>> a = 3
2.      >>> id(a)
3.      140710671365984
4.      >>> b = a
5.      >>> id(b)
6.      140710671365984
7.      >>> print(a,b,sep = ", ")
8.      3, 3
9.      >>> b = b + 5
10.     >>> id(b)
11.     140710671360907
12.     >>> print(a,b,sep = ", ")
13.     3, 8
```

Expliquemos qué es lo que ocurre por “debajo” en el ejemplo anterior:

- Línea 1: Python crea el objeto entero con valor 3 y lo asigna a una variable con nombre “a”.
- Línea 2-3: Vemos la dirección de memoria de este objeto entero.
- Línea 4: Python asigna el nombre de variable “b” al mismo objeto entero 3.
- Línea 5-6: Vemos que la variable “b” apunta a la misma dirección que la variable “a”.
- Línea 7-8: Se imprimen por pantalla ambas variables que están asignadas al mismo objeto. Obtenemos los mismos valores.
- Línea 9-10-11: Al intentar modificar un objeto inmutable Python crea un nuevo objeto. Cuando una variable está asociada a un objeto inmutable, al modificar la variable no se modifica el objeto, sino que la variable se asocia a un nuevo objeto. En este caso particular, Python crea el objeto entero 8 y lo asigna a la variable “b”. Vemos que ahora la dirección de memoria a la que apunta la variable “b” es diferente.
- Línea 12-13 Se imprimen nuevamente las dos variables y encontramos que cada una apunta a un objeto distinto.

Veamos ahora el comportamiento que tienen las variables que se asocian a objetos mutables:

```
1.      >>> a = [1,2,3]
2.      >>> id(a)
3.      140710671365983
4.      >>> b = a
5.      >>> id(b)
6.      140710671365983
7.      >>> (a,b)
8.      ([1, 2, 3], [1, 2, 3])
9.      >>> del b[0]
10.     >>> id(b)
11.     140710671365983
12.     >>> (a,b)
13.     ([2, 3], [2, 3])
14.
```

Explicando en detalle cada una de las líneas del ejemplo anterior:

- Línea 1: Python crea el objeto lista con valor [1,2,3] y lo asigna a una variable con nombre “a”.
- Línea 2-3: Vemos la dirección que ocupa el objeto lista.
- Línea 4: Python asigna el nombre de variable “b” al mismo objeto lista con valor [1,2,3].
- Línea 5-6: Vemos la dirección que ocupa el objeto lista.
- Líneas 7-8: Se imprimen por pantalla ambas variables que estan asignadas al mismo objeto. Obtenemos los mismos valores.
- Línea 9-10-11: Cuando una variable está asociada a un objeto mutable, al modificar la variable se modifica el objeto (en la mayoría de los casos). Es decir, se borra el elemento con índice 0 de la variable b que apunta a la lista [1,2,3] y puesto que esta variable es mutable, no se crea un nuevo objeto, sino que se altera el mismo. Vemos que la dirección de memoria al que apunta la variable “b” sigue siendo la misma.
- Línea 12-13: Se imprimen nuevamente las dos variables y encontramos que tanto el nombre “a” como el nombre “b” apuntan al mismo objeto modificado.

La diferencia entre intentar alterar un objeto inmutable y uno mutable se podría resumir de la siguiente manera:

- Al intentar modificar un objeto inmutable, se crea un nuevo objeto y se le asigna el nuevo nombre de variable.
- Al intentar modificar un objeto mutable, no se crea un objeto nuevo, sino que se modifica el existente, y consecuentemente, **todas las variables que apunten a ese objeto mostrarán ese cambio.**

Por lo tanto, aplicado al caso particular de las listas, cuando tenemos la siguiente asignación:

```
1.      >>> mi_lista1 = [1,2,3]
2.      >>> mi_lista2 = mi_lista1
3.
```

Esto no significa que estamos creando una nueva variable llamada mi_lista2 que es una copia de la variable mi_lista1, sino que la variable mi_lista2 está apuntando al mismo objeto que la variable mi_lista1 y consecuentemente, puesto que las listas son mutables, alterar una de ellas significa alterar el mismo objeto y cuando invoquemos cualquiera de las dos, la modificación será visible, tal y como se ha visto en el ejemplo previo.

En el caso de que nos interesara crear una nueva variable (mi_lista2) que apuntara a un objeto distinto pero que tuviera los mismos contenidos que mi_lista1, deberemos hacer uso de los slices tal y como vemos a continuación

```
1.      >>> mi_lista1 = [1,2,3]
2.      >>> mi_lista2 = mi_lista1[:]
3.      >>> del mi_lista2[0]
4.      >>> (mi_lista1, mi_lista2)
5.      ([1, 2, 3], [2, 3])
6.
```

Donde observamos que al borrar el elemento con índice cero de la lista mi_lista2 no afecta a la variable mi_lista1, ya que los objetos a los que cada una de las variables apuntan no son las mismas.

3. TUPLAS

Pasemos ahora a hablar de otro tipo de variables. Las tuplas.

Este tipo de variables, tiene muchas similitudes con las listas, son bastante recurrentes en el examen del PCAP y es por ello que repasaremos brevemente sus características principales

Una tupla es un **objeto inmutable** que se utiliza, al igual que las listas, para almacenar valores en su interior **de manera ordenada**. Podríamos decir, por tanto, que las tuplas son "listas inmutables". Existen dos maneras de asignar un objeto de tipo tupla a una variable en Python:

mi_tupla = (valor1, valor2,…., valorN)

mi_tupla = valor1, valor2,…., valorN

```
1.      >>> tupla1 = (1,"dos",3,True)
2.      >>> tupla2 = 1,False,5,9
3.
```

Una particularidad que tiene la asignación de las tuplas es que en el caso de querer una variable con una tupla de un solo valor en su interior tiene que asignarse de la siguiente manera:

```
1.      >>> tupla3 = (1,)
2.      >>> tupla4 = 1,
3.
```

Ya que, si no escribiésemos la "," después del valor, Python entendería que la variable apunta a un objeto de tipo entero.

La diferencia fundamental de las tuplas en comparación a las listas es su inmutabilidad. Si intentáramos modificar una obtendríamos el siguiente mensaje de error.

```
>>> tupla1 = (1,"dos",3,True)
>>> tupla1[0] = 4
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item
     assignment
>>> del tupla1[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object doesn't support item
     Deletion
```

Sin embargo, en todo lo relativo a lectura de la información que contienen este tipo de variables, todo lo visto para las listas funcionará igualmente para las tuplas.

Podemos indexar sin problema cualquier elemento de la tupla con las mismas condiciones que con las listas.

- Los índices deben ser enteros y pueden ser resultado de una expresión. En el caso de que sean de tipo float, devolverá un error.
- Los índices también pueden ser de signo negativo. La cota inferior y superior de los índices posibles viene determinado por la longitud total de la tupla con la misma expresión que con las listas:

-len(mi_tupla)<=índice<=len(mi_tupla) - 1

```
>>> tupla1 = (1,"dos",3,True)
>>> tupla1[2] # Índice positivo
3
>>> tupla1[-1]  # Índice negativo
True
>>> tupla1[1+2] # Índice como resultado de una
                # expresión
True
>>> tupla1[5//2] # Índice como resultado de una
                 # expresión
 3
>>> tupla1[5/2] # Los índices flotantes arrojan
                # error
Traceback (most recent call last):
   File "<stdin>", line 1, in <module>
TypeError: tuple indices must be
integers or slices, not float
```

```
>>> tupla1[10] #Índices fuera de rango arrojan error

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: tuple index out of range

```

Sin embargo, sí es posible utilizar la instrucción “del” para borrar la tupla completa. Por supuesto, después de borrarla, intentar acceder a ella nos devolverá un error.

```
>>> tupla1 = (1,"dos",3,True)
>>> del tupla1
>>> tupla1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'tupla1' is not defined

```

La función “len()” también funciona sin problema con las tuplas

```
>>> tupla1 = (1,"dos",3,True)
>>> len(tupla1)
4

```

De la misma manera podemos utilizar los slices

```
>>> tupla2 = (1, "dos",
               "otra cadena", False, [3,2],8)
>>> tupla2[2:]
('otra cadena', False, [3, 2], 8)
>>> tupla2[:4]
(1, 'dos', 'otra cadena', False)
>>> tupla2[1:-1:3]
('dos', [3, 2])
>>> tupla2[-1:0:-2]
(8, False, 'dos')

```

4. DICCIONARIOS

A continuación, hablaremos de otro tipo de objetos que es muy utilizado para almacenar datos. Los diccionarios.

5.1. Definición y principales características

Un diccionario es un **objeto mutable** que, a diferencia de las listas y tuplas, almacena datos de manera **no ordenada**, aunque desde Python 3.7 los diccionarios mantienen el orden. Cada elemento que los compone está formado por un par "clave-valor" y la manera de definirlos es la siguiente.

```
>>> diccionario = {"clave1":"valor1",
                   "clave2":"valor2",
                   "clave3":"valor3"}
```

Hablemos a continuación de los aspectos más relevantes a la hora de trabajar con diccionarios.

- Los diccionarios **no pueden tener claves repetidas**.
- El diccionario, como podemos intuir por su nombre, funciona de manera muy similar a los tradicionales diccionarios que conocemos, donde a cada palabra le asignamos una definición. La idea es la misma, a cada "clave" se le asigna un "valor" y si en un futuro necesitamos recuperar el valor de determinada clave lo haremos a través de la siguiente sentencia

```
>>> diccionario["clave1"]
'valor1'
```

- Las claves y valores pueden ser de cualquier tipo.

```
>>> dict = {1:1,
            "cadena":2,
            (1,2):"texto",
             True:[1,2]}
```

- Es posible utilizar la función len() también con los diccionarios.

5.2. Lectura y modificación de diccionarios

Además de recuperar el valor de una clave a través de la misma como se ha visto anteriormente, existen algunos métodos que pueden resultar de mucha utilidad a la hora de intentar leer todas las claves, o recuperar todos los valores o los pares clave-valor en una lista.

Estos métodos son: **keys(), values() y items()**.

Método keys()

Al invocar este metodo sobre un diccionario, obtendremos como resultado una lista con todas las claves que componen al diccionario. Un ejemplo sería el siguiente.

```
1.      >>> dict = {1:1, "cadena":2,
2.                   (1,2):"texto", True:[1,2]}
3.      >>> print(dict.keys())
4.      dict_keys([1, 'cadena', (1, 2)])
5.
```

Este método resulta muy útil, ya que se combina con el bucle for para poder recorrer todos los elementos del diccionario.

Método values()

Este método es el homólogo del método keys() pero se encarga de devolver una lista con todos los valores de cada una de las claves. Dejamos un ejemplo a continuación.

```
1.      >>> dict = {1:1, "cadena":2,
2.                   (1,2):"texto", True:[1,2]}
3.      >>> print(dict.values())
4.      dict_values([[1, 2], 2, 'texto'])
5.
```

Método items()

Al invocarse sobre un diccionario, el método items() devuelve una **lista de tuplas**, donde cada elemento contendrá tanto la clave como el valor asignado a la misma.

```
>>> dict = {1:1, "cadena":2,
            (1,2):"texto", True:[1,2]}
>>> print(dict.items())
dict_items([(1, [1, 2]), ('cadena', 2),
    ((1, 2), 'texto')])
```

Como hemos mencionado en la definición de los diccionarios, estos objetos son de tipo mutable, con lo cual tiene sentido que ahora hablemos brevemente de los procedimientos mediante los cuales se puede tanto añadir, actualizar como eliminar elementos de los mismos.

Añadir o actualizar elementos en los diccionarios

El procedimiento que se sigue tanto para añadir como para actualizar elementos en un diccionario es muy similar a cómo se hacia en las listas, salvo que en esta ocasión, en vez de utilizar los índices, utilizaremos las "claves" para ello. De tal manera que si queremos añadir un nuevo elemento a un diccionario debemos hacer lo siguiente

```
>>> dict = {1:1, "cadena":2,
            (1,2):"texto", True:[1,2]}
>>> dict["nueva_clave"] = "nuevo_valor" #Introducimos
                        #un nuevo par clave-valor
>>> dict
{1: [1, 2], 'cadena': 2,
(1, 2): 'texto', 'nueva_clave': 'nuevo_valor'}
```

En el caso de que la clave ya exista, invocar la sentencia anterior sobreescribirá el valor asignado a la clave actualizando el valor de la misma.

```
>>> dict = {1:1, "cadena":2,
            (1,2):"texto", True:[1,2]}
>>> dict["cadena"] = "valor_actualizado"
      #Actualizamos el valor de la clave "cadena"
>>> dict
{1: [1, 2], 'cadena': 'valor_actualizado',
(1, 2): 'texto'}

```

Borrar elementos en los diccionarios

Finalmente, tenemos la posibilidad de eliminar elementos de un diccionario. También se logra mediante la instrucción "del". Nuevamente, muy similar al caso de las listas, pero utilizando las claves en vez de los índices.

```
>>> dict = {1:1, "clave2":2,
             "clave3":"texto","clave4":[1,2]}
>>> del dict["clave3"]
>>> dict
{1: 1, 'clave2': 2, 'clave4': [1, 2]}
>>> del dict["clave7"]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'clave7'

```

Por supuesto, intentar borrar una clave que no exista arroja un error, cuidado con eso.

5. Iterar a través de las Listas, Tuplas y Diccionarios

Finalmente, para concluir con la lección imaginemos que tuviésemos que recorrer todos los elementos de una lista, una tupla o un diccionario, por ejemplo, para hacer una operación sobre ellos como mostrarlos de uno en uno por pantalla. Operación muy útil y utilizada de forma recurrente en muchos de nuestros programas. Para ello Python permite **iterar** a través de ellos de forma muy sencilla a través de la instrucción **for**.

En el siguiente ejemplo se muestra cómo se podría iterar a través de una lista:

```
1.      >>> mi_lista = ["Lola", "Pepe", "Juan", "Maria"]
2.      >>> for elemento in mi_lista:
3.      >>>     print("Nombre: ", elemento)
4.      Nombre: Lola
5.      Nombre: Pepe
6.      Nombre: Juan
7.      Nombre: Maria
8.
```

Para el caso de las tuplas sería similar:

```
1.      >>> mi_tupla = ("Lola", "Pepe", "Juan", "Maria")
2.      >>> for elemento in mi_tupla:
3.      >>>     print("Nombre: ", elemento)
4.      Nombre: Lola
5.      Nombre: Pepe
6.      Nombre: Juan
7.      Nombre: Maria
8.
```

En el caso de los diccionarios, si hacemos un bucle similar al visto en los ejemplos anteriores, la variable elemento iría tomando el valor de las diferentes claves:

```
1.      >>> mi_dict = {"Clave1": "Valor1",
2.                     "Clave2": "Valor2"}
3.      >>> for elemento in mi_dict:
4.      >>>     print("Soy: ", elemento)
5.      Soy: Clave1
6.      Soy: Clave2
7.
```

De esta forma si quisiésemos mostrar también el valor, podríamos obtenerlo de la siguiente manera:

```
>>> mi_dict = {"Clave1": "Valor1",
                "Clave2": "Valor2"}
>>> for elemento in mi_dict:
>>>     print("Soy: ", elemento)
>>>     print("Mi valor es: ", mi_dict[elemento])
Soy: Clave1
Mi valor es: Valor1
Soy: Clave2
Mi valor es: Valor2
```

Otra opción podría ser utilizar la función ítems() que al devolver una lista de tuplas con clave y valor no permitiría definir el siguiente bucle:

```
>>> mi_dict = {"Clave1": "Valor1",
                "Clave2": "Valor2"}
>>> for clave, valor in mi_dict:
>>>     print("Soy: ", clave)
>>>     print("Mi valor es: ", valor)
Soy: Clave1
Mi valor es: Valor1
Soy: Clave2
Mi valor es: Valor2
```

6. Puntos Clave

Finalmente, resumiremos los principales aspectos de los contenidos repasados en esta lección en los siguientes puntos:

- Hemos estudiado las estructuras principales de almacenamiento de datos en Python: listas, tuplas y diccionarios.
- Hemos explicado las principales características de cada una de estos objetos, desde la sintaxis de su asignación a variables como los métodos de lectura de los datos contenidos en ellas y los procedimientos para modificar aquellas que sí son modificables.
- Hemos clasificado estas estructuras en dos grandes grupos: **mutables** e **inmutables.** Hemos explorado el significado e implicaciones de esta clasificación.
- Hemos visto como iterar a través de estas estructuras para poder trabajar elemento a elemento.